Functional Dependencies and Normalization

- There are many forms of constraints on relational database schemata other than key dependencies.
- Undoubtedly most important is the *functional dependency.*
- A functional dependency, or *FD,* is a constraint on a single relation schema.
- Basically, it is a key constraint on a subset of the set of all attributes.

Formally:

Definition: Let R[$\mathbf{A}$] be a relation schema, and let X, Y $\subseteq \mathbf{A}$. The constraint $X \rightarrow Y$ is defined as follows:

For any tuples $t_1[\mathbf{A}]$, $t_2[\mathbf{A}]$,

$$t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

Observe: The functional dependency $X \rightarrow Y$ is satisfied iff X is a superkey for $\pi_{X \cup Y}(R[\mathbf{A}])$.

- In words, the FD $X \rightarrow Y$ is satisfied iff X is a superkey for the projection onto the attributes $X \cup Y$.

Example:

Rail_Schedule				
Engineer	Train	Date	Departure	Platform

{Train} $\rightarrow$ {Departure}
{Engineer, Date, Departure} $\rightarrow$ {Train}
{Train, Date} $\rightarrow$ {Engineer, Platform}

Semantic Consequence:

It is often the case that if certain FD's hold, then others must hold as well.

Examples:

If {Train, Date} $\rightarrow$ {Engineer, Platform} and
{Engineer, Date, Departure} $\rightarrow$ {Train} hold
then so does
{Engineer, Date, Departure} $\rightarrow$ {Platform}.

If {Train} $\rightarrow$ {Departure} holds
then so does
{Train, Engineer} $\rightarrow$ {Departure}.

Formally, let **A** be a set of attributes, and let $F_1, F_2, .., F_n$, and G be FD's over **A**. G is a *semantic consequence* of $\{F_1, F_2, .., F_n\}$, written

$$\{F_1, F_2, .., F_n\} \models G$$

if the FD G holds whenever all of the elements of $\{F_1, F_2, .., F_n\}$ do. (This applies to any relation whatever on the attribute set **A**.)
If $\{G_1, G_2, .., G_m\}$ is a set of FD's, then

$$\{F_1, F_2, .., F_n\} \models \{G_1, G_2, .., G_m\}$$

means that $\{F_1, F_2, .., F_n\} \models G_i$ for each i, $1 \leq i \leq m$.

Examples:

{{Train, Date} → {Engineer, Platform}
{Engineer, Date, Departure} → {Train}}
⊨
{Engineer, Date, Departure} → {Platform}

{{Train} → Departure}}
⊨ {Train, Engineer} → {Departure}.

Closure: The set G of all FD's for which $\{F_1, F_2, .., F_n\} \models G$ holds is called the *(semantic) closure* of $\{F_1, F_2, .., F_n\}$, and is denoted $\{F_1, F_2, .., F_n\}^+$. Two sets of FD's $\mathcal{F}$ and $\mathcal{G}$ are said to be *equivalent* if $\mathcal{F}^+ = \mathcal{G}^+$.

Exterior: The set G of all FD's for which $\{F_1, F_2, .., F_n\} \models G$ does not hold, written

$$\{F_1, F_2, .., F_n\} \not\models G$$

is called the *(semantic) exterior* of $\{F_1, F_2, .., F_n\}$, and is denoted $\{F_1, F_2, .., F_n\}^-$.

Example:

{{Train} → {Departure}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer, Platform}
} ⊭
{Departure} → {Train}.

Question: How is the relation $\models$ determined?

- Purely semantic approaches are possible, but impractical. (Compare truth tables in propositional logic.)

- Syntactic inference system: A *syntactic inference system* is a collection of rules which allows us to conclude new assertions (*i.e.,* FD's) from existing ones.

Example: FD's obey a transitive rule. If $A \rightarrow B$ and $B \rightarrow C$ both hold (for any attributes A, B, and C whatever), then so too does $A \rightarrow C$. Thus, we might include such a rule in a syntactic system.

For an inference system, the symbol which is typically used is $\vdash$. Thus,

$$\{F_1, F_2, .., F_n\} \vdash G$$

means that G can be deduced from $\{F_1, F_2, .., F_n\}$ by application of the extant system of syntactic rules. (Alternately, there is a *proof* of G from $\{F_1, F_2, .., F_n\}$.)

Example: A possible rule is

$$\{ A \rightarrow B, B \rightarrow C\} \vdash A \rightarrow C.$$

A Syntactic Proof System:

In the following rules, let **A** be a set of attributes, and let W, X, Y, and Z be arbitrary subsets of **A**.

A1 (Reflexivity): $\{ Y \subseteq X \} \vdash X \rightarrow Y$.

A2 (Augmentation): $\{ X \rightarrow Y \} \vdash X \cup Z \rightarrow Y \cup Z$.

A3 (Additivity) : $\{ X \rightarrow Y, X \rightarrow Z \} \vdash X \rightarrow Y \cup Z$.

A4 (Projectivity): $\{ X \rightarrow Y \cup Z \} \vdash X \rightarrow Y$.

A5 (Transitivity): $\{ X \rightarrow Y, Y \rightarrow Z \} \vdash X \rightarrow Z$.

A6 (Pseudotransitivity):

$$\{ X \rightarrow Y, Y \cup Z \rightarrow W \} \vdash X \cup Z \rightarrow W.$$

Contrary to the assertion in the text, we cannot "prove" these rules. However, it can be established that they have certain fundamental logical properties. But first, we must be clear about what is meant by an inference.

Definition: Let $\vdash$ be an inference relation, and let $\varphi_1, \varphi_2, ..., \varphi_n$, and ψ be assertions. A *proof* of ψ from $\{\varphi_1, \varphi_2, ..., \varphi_n\}$ is a sequence of assertions

$$\zeta_1, \zeta_2, .., \zeta_i, \zeta_{i+1}, .., \zeta_{k-1}, \zeta_k$$

with the property that

1. The final element in the list is the conclusion which is sought; *i.e.,* $\zeta_k = \psi$.

2. Every element in the list is either one of the φ_l's or else a consequence, via $\vdash$, of some of the preceding elements in the list. Formally, for each i, $1 \leq i \leqslant k$, either $\zeta_i \in \{\varphi_1, \varphi_2, ..., \varphi_n\}$ or else $Z \vdash \zeta_i$ for some $Z \subseteq \{\zeta_1, \zeta_2, .., \zeta_{i-1}\}$.

Example: Prove
{Engineer, Date, Departure} $\rightarrow$ {Platform}
from the axioms of the earlier example.

1. {Train} $\rightarrow$ {Departure}
   (Given)

2. {Engineer, Date, Departure} $\rightarrow$ {Train}
   (Given)

3. {Train, Date} $\rightarrow$ {Engineer, Platform}
   (Given)

4. {Engineer, Date, Departure} $\rightarrow$ {Train, Date}
   (A2: Augmentation of 2)

5. {Engineer, Date, Departure} $\rightarrow$ {Engineer, Platform}
   (A5: Transitivity on 4 and 3)

6. {Engineer, Date, Departure} $\rightarrow$ {Platform}.
   (A4: Projectivity on 5)

Formal properties of inference systems:

- *Soundness:* Everything which can be proven is true. Formally, If $\{\varphi_1, \varphi_2, ..., \varphi_n\} \vdash \psi$, then $\{\varphi_1, \varphi_2, ..., \varphi_n\} \models \psi$.

- *Completeness:* Everything which is true can be proven. Formally, if $\{\varphi_1, \varphi_2, ..., \varphi_n\} \models \psi$, then $\{\varphi_1, \varphi_2, ..., \varphi_n\} \vdash \psi$.

- *Decidability*: There is an algorithm which can apply the proof rules and determine whether or not there is a proof of the desired conclusion from the axioms. (The process cannot loop forever in a search.)

Fact: The axioms A1-A6 are sound and complete, and possess a decidable inference algorithm.

(It is easy to see that things are decidable in this context. Why?)

Fact: The subset consisting of just A1 (Reflexivity), A2 (Augmentation), and A6 (Pseudotransitivity) is complete.

A more intuitive inference system for FD's:

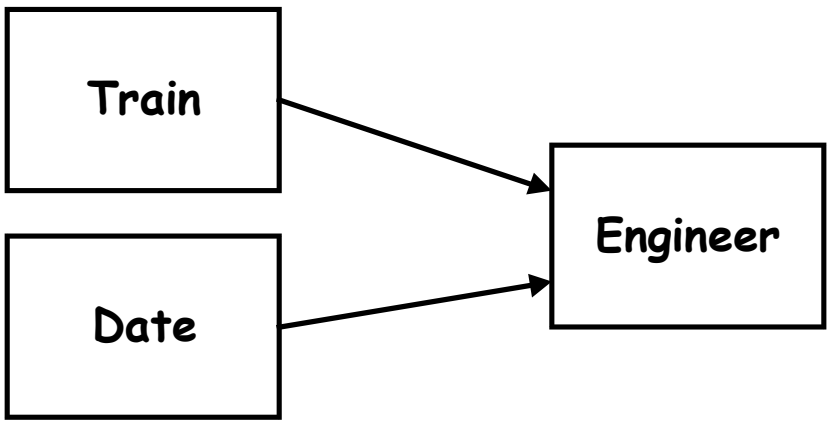
The following system is based upon directed acyclic graphs (DAG):

- Reduce all FD's to those with only one attribute on the right-hand side (RHS).

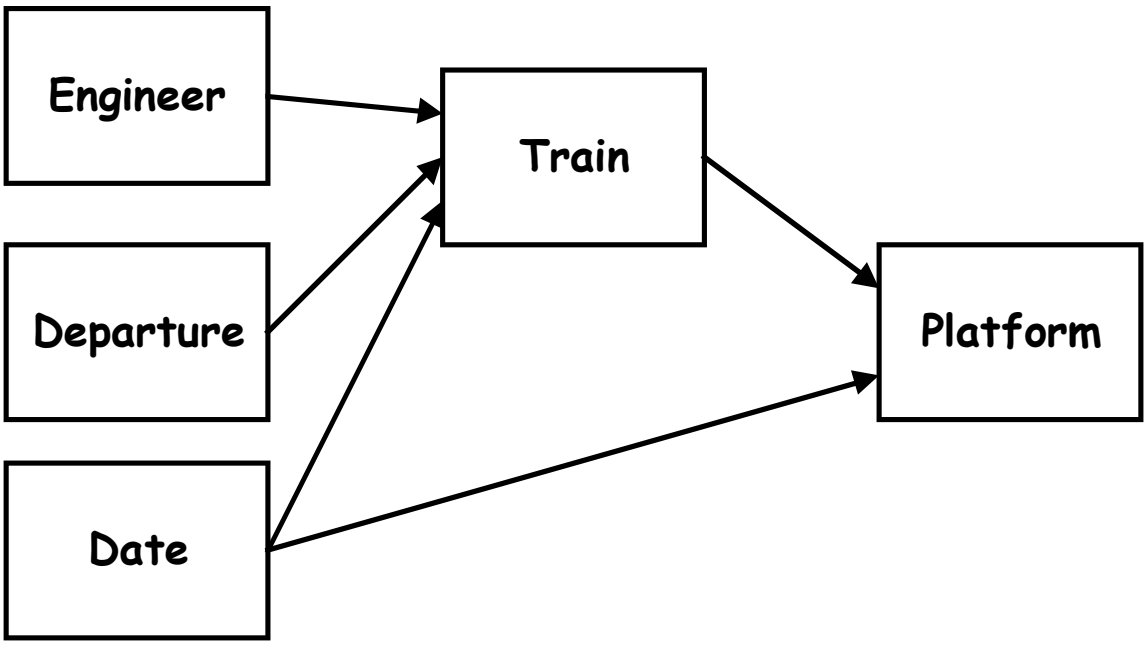
Example: {Train, Date} → {Engineer, Platform} becomes:
{Train, Date} → {Engineer}
{Train, Date} → { Platform}
Clearly, these two FD's are equivalent to the one above. Call such an FD *simple.*

- Represent each simple FD by a DAG in which the nodes are attributes and the edges run from left-hand side (LHS) attributes to right-hand side (RHS) attributes.

Example: The FD {Train, Date} → {Engineer} is represented as follows.



Derivations are represented by gluing these graphs together. For example, here is a derivation of {Engineer, Date, Departure} $\rightarrow$ {Platform}.



This graph embodies the FD's
{Engineer, Date, Departure} $\rightarrow$ {Train}
{Train, Date} $\rightarrow$ {Platform}
as axioms. It derives the FD
{Engineer, Date, Departure} $\rightarrow$ {Platform}
because

- The LHS attributes are the initial nodes of the graph.
- All RHS attributes are named in nodes which are connected to these initial nodes.

Fact: This DAG procedure is sound and complete for inference on FD's. (D. Maier, *The Theory of Relational Databases,* Computer Science Press, 1983.)

Normalization:

- Schemata constrained by arbitrary sets of FD's have certain *anomalies* which make them undesirable. To remedy this situation, there have been quite a number of *normal forms* proposed which deal with these issues.

- There are two approaches to normalization:

  - In the *decomposition approach*, one starts with a relational database schema with perhaps only one or a very few relations, and decomposes the relations (using projection) into smaller ones in an effort to remove the problems.

  - In the *synthetic approach*, one starts with a set of FD's, and then attempts to construct a relational database schema which embodies those FD's while avoiding any anomalies.

- Each approach as its advantages and disadvantages.

- Both approaches will be examined in these slides.

- First, we need to consider some normal forms.

Normal Forms:

- In early papers, E. F. Codd, who is credited with “inventing” the relational model, introduced three normal forms.

First Normal Form:

This just says that domains consist of atomic values, and may not themselves be structured. In most modern work, this property is built into the model.

Second Normal Form:

Consider a slight modification of the Rail Schedule example.

Rail_Schedule				
Engineer	Train	Date	Departure	Platform

{Train} → {Departure, Platform}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer}

A train now must depart from the same platform every day. There is a so-called anomaly in this schema.

- Note that the values of Departure and Platform are determined by the value of the attribute Train alone, and so many tuples replicate this information.

Rail_Schedule				
Engineer	Train	Date	Departure	Platform
Ola	12	23	0800	4
Kari	12	24	0800	4
Ola	12	25	0800	4
Renée	13	23	0930	5
René	13	24	0930	5
Renée	13	25	0930	5

- If the platform or departure time for a train are to be changed, they must be changed in every tuple associated with that train. (Update anomaly)

- To insert a new train, we must have information on a date for that train, and for an engineer for each such date. (Insertion anomaly)

- If no information for any date is available for a train, its departure and platform information are lost. (Deletion anomaly)

- These problems may be (partially) remedied by decomposing the relation into two pieces.

- In the following solution, the single relation is broken into two as follows. Now, the update anomalies have disappeared.

Train	Departure	Platform
12	0800	4
13	0930	5

Engineer	Train	Date
Ola	12	23
Kari	12	24
Ola	12	25
Renée	13	23
René	13	24
Renée	13	25

Formalization of 2NF:

Let R[**A**] be a relation schema, and let $\mathcal{F}$ be a set of FD's on R[**A**]. Assume, without loss of generality, that all FD's in $\mathcal{F}$ have only a single element on the RHS. Then $\mathcal{F}$ is said to be in 2NF if for each FD $X \rightarrow \{B\} \in \mathcal{F}$ with $B \notin X$, the following condition is satisfied:

If
- There is a candidate key K such that $X \subseteq K$; and
- There is no candidate key L such that $B \in L$ (*i.e.,* B is not a *prime attribute*);

Then
- X = K.

- In this case, it is said that B is *fully dependent*, or *irreducibly dependent,* upon each candidate key, because there can be no proper subset of such a key upon which B depends.

- So, a schema is in 2NF if each nonprime attribute is fully dependent upon each candidate key.

- The original single-relation schema violates 2NF.

- The candidate keys of the original single-relation schema are
  {Train, Date} and {Engineer, Date, Departure}.

- The dependency {Train} → {Platform} violates the conditions of 2NF, since Train is a prime attribute which is not a candidate key, while Platform is not a prime attribute.

- Verify that the decomposed schema is in 2NF:

{Train} → {Departure, Platform}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer}

Train	Departure	Platform

Engineer	Train	Date

- This is trivial since each relation contains only one of the original FDs, and the left-hand side of each of those FDs is a key for that relation.

- Note that the second FD is no longer recaptured by a relation! (The decomposition is not *dependency preserving.* More on this later.)

Here is an alternate solution which is also in 2NF:

Train	Departure	Platform
12	0800	4
13	0930	5

Engineer	Train	Date	Departure
Ola	12	23	0800
Kari	12	24	0800
Ola	12	25	0800
Renée	13	23	0930
René	13	24	0930
Renée	13	25	0930

- Note that an update anomaly remains.

Verify that it is in 2NF:

{Train} → {Departure, Platform}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer}

Train	Departure	Platform

Engineer	Train	Date	Departure

- The first relation is trivially in 2NF, since {Train} → {Departure, Platform} is its only FD.
- In the second relation, the candidate keys are {Engineer, Date, Departure} and {Train, Date}.
- Note that all attributes in the second relation are prime; therefore, it must be in 2NF.
- Note that each FD is embodied in at least one of the relations. (The decomposition is *dependency preserving.* More on this later.)

Third Normal Form:

Now let us modify the schema a bit further, adding a new attribute Loco (for Locomotive), and add the assumption that an engineer always drives the same locomotive, and a locomotive is only used for one train. The constraints then become:

Rail_Schedule					
Engineer	Loco	Train	Date	Departure	Platform

{Train} → {Departure, Platform}
{Engineer} → {Loco}
{Loco} → {Train}
{Train, Date} → {Engineer}

Example instance:

Rail_Schedule					
Engineer	Loco	Train	Date	Departure	Platform
Ola	A12	12	23	0800	4
Kari	A12	12	24	0800	4
Ola	A12	12	25	0800	4
Renée	A22	13	23	0930	5
René	A22	13	24	0930	5
Renée	A22	13	25	0930	5

- This schema is not even in 2NF.
    - The candidate keys are {Train, Date}, {Loco, Date}, and {Engineer, Date}.
    - Platform is not a prime attribute, so {Train} $\rightarrow$ {Platform} violates 2NF.

The decomposition

Train	Departure	Platform
12	0800	4
13	0930	5

Engineer	Loco	Train
Ola	A12	12
Kari	A12	12
Renée	A22	13
René	A22	13

Engineer	Train	Date
Ola	12	23
Kari	12	24
Ola	12	25
Renée	13	23
René	13	24
Renée	13	25

is easily verified to be in 2NF.

- Yet, there is an anomaly because of the $\{Loco\} \rightarrow \{Train\}$ relationship in the second relation.

- A schema is said to be in *third normal form (3NF)* with respect to a set $\mathcal{F}$ of FD's if for every $X \rightarrow Y \in \mathcal{F}$ with $Y \not\subseteq X$, either X is a superkey, or else $Y \setminus X$ consists entirely of prime attributes.

- Equivalently, in a form which parallels that for 2NF:

Let R[**A**] be a relation schema, and let $\mathcal{F}$ be a set of FD's on R[**A**]. Assume, without loss of generality, that all FD's in $\mathcal{F}$ have only a single element on the RHS. Then $\mathcal{F}$ is said to be in 3NF if for each FD $X \rightarrow \{B\} \in \mathcal{F}$ with $B \notin X$, the following condition is satisfied:

If

- ~~There is a candidate key K such that $X \subseteq K$; and~~
- There is no candidate key L such that $B \in L$ (*i.e.,* B is not a *prime attribute*);

Then

- X ~~= K~~ is a superkey.

.

- Note that {Loco} $\rightarrow$ {Train} violates this condition in the second relation of the decomposition, since Engineer is the only key.

- The following decomposition resolves the problem, providing a 3NF decomposition.

Train	Departure	Platform
12	0800	4
13	0930	5

Engineer	Train	Date
Ola	12	23
Kari	12	24
Ola	12	25
Renée	13	23
René	13	24
Renée	13	25

Engineer	Loco
Ola	A12
Kari	A12
Renée	A22
René	A22

Loco	Train
A12	12
A22	13

Fact: 3NF $\Rightarrow$ 2NF (trivially).

Why is 3NF strictly stronger than 2NF?

- In 2NF, it is possible to have an FD $X \rightarrow A$ in which none of the elements of X nor the attribute A are members of any candidate key.
- In 3NF, if X is not a superkey, then A must be a member of a candidate key.

We can verify that, for the example, all dependencies are embodied in one of the relations, so the decomposition is dependency preserving.

{Train} $\rightarrow$ {Departure, Platform}
{Engineer} $\rightarrow$ {Loco}
{Loco} $\rightarrow$ {Train}
{Train, Date} $\rightarrow$ {Engineer}

Train	Departure	Platform

Engineer	Train	Date

Engineer	Loco

Loco	Train

Boyce-Codd Normal Form:

Return to the example used in 2NF.

Rail_Schedule				
Engineer	Train	Date	Departure	Platform

{Train} → {Departure, Platform}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer}

Train	Departure	Platform
12	0800	4
13	0930	5

Engineer	Train	Date	Departure
Ola	12	23	0800
Kari	12	24	0800
Ola	12	25	0800
Renée	13	23	0930
René	13	24	0930
Renée	13	25	0930

- It is easy to see that this schema is in 3NF.
- {Train} → {Departure} does not violate 3NF in the second relation, since Departure is a prime attribute.

3NF says:

- If the RHS of an FD is not a member of a candidate key, then the LHS must be a superkey.

- If the RHS of an FD is a member of a candidate key, then there is no restriction on the LHS.

- Boyce-Codd normal form remedies this situation by strengthening the condition:

- A set of FD's is said to be in *Boyce-Codd normal form (BCNF)* with respect to a set $\mathcal{F}$ of FD's if for every $X \rightarrow Y \in \mathcal{F}$ with $Y \not\subseteq X$, X is a superkey, regardless of whether or not Y consists entirely of prime attributes.

- The example schema is not in BCNF.

- The first decomposed schema from the 2NF example is in BCNF:

{Train} → {Departure, Platform}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer}

Train	Departure	Platform

Engineer	Train	Date

- However, this decomposition is not *dependency preserving*, since the dependency
  {Engineer, Date, Departure} → {Train}
  is not embedded in any of the component schemata.

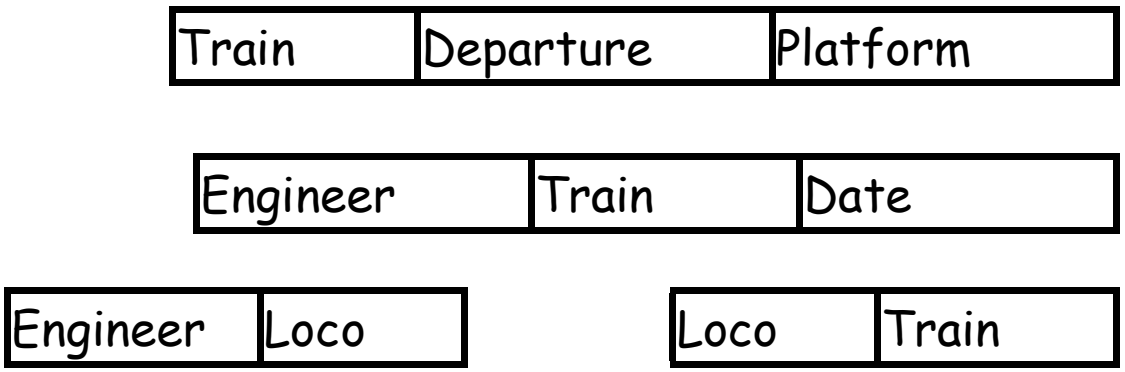
- In general, it is not possible to decompose a schema into BCNF without incurring this sort of loss of embedded dependencies.

- On the other hand, it is always possible to decompose a schema into 3NF without such loss.

- While BCNF is “better” than 3NF in terms of avoiding update anomalies, it is worse in that dependencies may be lost, or at least FD’s may need to be verified by checking several relations.

- Sometimes, BCNF can be achieved without loss of dependencies, as in the 3NF example given previously.

{Train} → {Departure, Platform}
{Engineer} → {Loco}
{Loco} → {Train}
{Train, Date} → {Engineer}



- The concepts of lossless and dependency-preserving decompositions are next examined more formally, and in greater detail.

Minimal covers for a set of FDs:

- A *cover* for a set $\mathcal{F}$ of FDs is a set C of FDs with the property that $C^+ = \mathcal{F}^+$.
- Informally, a cover C of $\mathcal{F}$ is *minimal* if it does not have any redundancy.
- There are several forms of redundancy.
- Example:

$\mathcal{F}$ = { A $\rightarrow$ BC, B $\rightarrow$ C, AB $\rightarrow$ D, AC $\rightarrow$ D }

- First reduce RHS to single attribute:
  FD: A $\rightarrow$ BC
  - May be replaced by {A $\rightarrow$ B, A $\rightarrow$ C}.

- Reducible LHS FD: AB $\rightarrow$ D
  - May be replaced by A $\rightarrow$ D since A $\rightarrow$ B.

- May remove implied FD: A $\rightarrow$ C
  - Implied by { A $\rightarrow$ B, B $\rightarrow$ C }.

- In each case, the remaining set of FDs has the same closure as the original set.

This process will now be illustrated more formally.

The algorithm to minimize a set $\mathcal{F}$ of FDs proceeds in three steps:

1. Decompose each FD with more than one attribute on the RHS to a set of FDs with the same LHS and exacly one attribute on the RHS.

2. Remove unnecessary attributes from the LHS of each FD. (May not be unique.)

3. For each remaining FD f in the set,
   if $\mathcal{F}^+ = \mathcal{F} \setminus \{ f \}^+$, then set $\mathcal{F} := \mathcal{F} \setminus \{ f \}$.
   (May not be unique.)

- The steps must be performed in this order.

Example from above:

$\mathcal{F} = \{ A \rightarrow BC,\ B \rightarrow C,\ AB \rightarrow D,\ AC \rightarrow D \}$

Step 1:

$\mathcal{F} = \{ A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C,\ AB \rightarrow D,\ AC \rightarrow D \}$

Step 2:

$\mathcal{F} = \{ A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C,\ A \rightarrow D,\ A \rightarrow D \} =$
$\{ A \rightarrow B,\ A \rightarrow C,\ B \rightarrow C,\ A \rightarrow D \}$

Step 3:

$\{ A \rightarrow B,\ B \rightarrow C \} \models A \rightarrow C$, so
$\mathcal{F} = \{ A \rightarrow B,\ A \rightarrow C,\ A \rightarrow D \}$

A more complex example:

$\mathcal{F}$ = { A $\rightarrow$ B, ABCD $\rightarrow$ E, EF $\rightarrow$ GH, ACDF$\rightarrow$ EG }

Step 1:
$\mathcal{F}$ = { A $\rightarrow$ B, ABCD $\rightarrow$ E, EF $\rightarrow$ G, EF $\rightarrow$ H,
ACDF$\rightarrow$ E, ACDF$\rightarrow$ G }

Step 2:
$\mathcal{F}$ = { A $\rightarrow$ B, ACD $\rightarrow$ E, EF $\rightarrow$ G, EF $\rightarrow$ H,
ACD$\rightarrow$ E, ACDF$\rightarrow$ G } =
{ A $\rightarrow$ B, ACD $\rightarrow$ E, EF $\rightarrow$ G, EF $\rightarrow$ H,
ACDF$\rightarrow$ G }

Step 3:
{ ACD $\rightarrow$ E, EF $\rightarrow$ G } $\models$ ACDF$\rightarrow$ G so
$\mathcal{F}$ = { A $\rightarrow$ B, ACD $\rightarrow$ E, EF $\rightarrow$ G, EF $\rightarrow$ H }

- In general, Steps 2 and 3 can be very complex, but they can usually be solved by inspection for small examples.

- This process is also described in Algorithm 10.2 of the textbook (Algorithm 14.2 in the third edition).

Dependency-Preserving Decompositions:

- Let R[**A**] be a relation schema, and let $\mathcal{F}$ be a set of FD's on **A**. Let **B** $\subseteq$ **A**. The *full projection* of $\mathcal{F}$ onto **B,** denoted $\pi_{\mathbf{B}}(\mathcal{F}^+)$, is the set of all elements $X \rightarrow Y \in \mathcal{F}^+$ for which $X \cup Y \subseteq$ **B**.

- Let R[**A**] be a relation schema, and let $\mathcal{F}$ be a set of FD's on **A**. Let

  $$D = \{R_1[\mathbf{A}_1], R_2[\mathbf{A}_2],.., R_n[\mathbf{A}_n]\}$$

  be a decomposition of R[**A**] into projections. The decomposition D is said to be *dependency preserving* if:

  $$(\pi_1(\mathcal{F}^+) \cup \pi_2(\mathcal{F}^+) \cup \ldots \cup \pi_n(\mathcal{F}^+))^+ = \mathcal{F}^+$$

  - The set $\mathcal{F}$ of FDs *embeds* in D if for each $X \rightarrow Y \in \mathcal{F}$, $X \cup Y \subseteq \mathbf{A}_i$ for some $i \in \{1,2,..n\}$.

  - To show that D is dependency preserving, it suffices to find a cover C of $\mathcal{F}$ which embeds in D.

- Re-examine the examples.

Finding a dependency-preserving 3NF decomposition of an arbitrary schema constrained by FD's:

Algorithm: Input: A single relation schema R[**A**], together with a set $\mathcal{F}$ of FD's on **A**.

1. Find a minimal cover C for $\mathcal{F}$.

2. Group the FD's in C by LHS.

$G_1 = \{X_1 \rightarrow A_{1j} \mid 1 \leq j \leq m_1\}$
$G_2 = \{X_2 \rightarrow A_{2j} \mid 1 \leq j \leq m_2\}$
$\vdots$
$G_n = \{X_n \rightarrow A_{nj} \mid 1 \leq j \leq m_n\}$

3. For each group $G_i = \{X_I \rightarrow A_{ij} \mid 1 \leq j \leq m_i\}$, let

$$Y_i = X_i \cup \{ A_{ij} \mid 1 \leq j \leq m_i\}.$$

Include in the decomposition the relation schema π[R], together with the constraints in $\pi(\mathcal{F}^+)$.

4. If $Y_1 \cup Y_2 \cup .. \cup Y_n \neq$ **A**, include a relation schema, with no nontrivial dependencies, on attributes **A** \ $(Y_1 \cup Y_2 \cup .. \cup Y_n)$.

- This is actually a *synthesis* algorithm, in that the schema is built directly from the FDs. Only step 4 requires that the total set of attributes be known.

- It is clear that this produces a dependency-preserving schema, since every FD is embodied in one of the constructed relations.

- It is less clear that this produces a 3NF decomposition; this will not be proven here.

- If there is already more than one relation in the schema to be decomposed, decompose each separately. As long as the input is a 3NF schema, so too will be the output.

- Examples: The running examples of these slides are fairly trivial for this algorithm, since the dependencies already form a minimal cover of themselves, and each has a distinct LHS.

Other examples:

Consider the very simple example R[ABC], with $\mathcal{F}$ = {A $\rightarrow$ B, B $\rightarrow$ C, A $\rightarrow$ C }.

- The full schema is not in 3NF, since B is not a superkey and C is not prime. (Hence B $\rightarrow$ C is a "problem" dependency.)

- The decomposition algorithm produces:

  R_1[AB] with FD set {A $\rightarrow$ B} and
  R_2[BC] with FD set {B $\rightarrow$ C}.

- Note that the original set of dependencies is not a minimal cover of itself; A $\rightarrow$ C is redundant. If we had included A $\rightarrow$ C in the algorithm, the (non-3NF) decomposition

  R_1[ABC] with FD set {A $\rightarrow$ B, A $\rightarrow$ C} and
  R_2[BC] with FD set {B $\rightarrow$ C}

  would have been obtained.

- Thus, the process of finding a minimal cover is essential.

Consider next the simple example R[ABC] with $\mathcal{F}$ = {A $\rightarrow$ B}.

- The algorithm constructs the schema

  R_1[AB] with FD set {A $\rightarrow$ B},

- It then adds the schema

  R_2[C] with empty FD set $\varnothing$.

  to achieve attribute preservation.

- This is a “bad” decomposition, since C may depend upon A and B in other ways. There is no way that we can recover the original relation on R[ABC] from the projections R_1[AB] and R_2[C].

- This is not the case with the previous example; there the original relation could be recovered from the projections.

Lossless Decompositions:

- Let R[**A**] be a relation schema, and let $\mathcal{F}$ be a set of FD's on **A**. Let

  $$D = \{R_1[\mathbf{A}_1], R_2[\mathbf{A}_2], .., R_n[\mathbf{A}_n]\}$$

  be a decomposition of R[**A**] into projections. The decomposition D is said to be *lossless* if for any relation $r \in Sat(R[\mathbf{A}], \mathcal{F})$,

  $$\pi_1(r) \bowtie \pi_2(r) \bowtie \ldots \bowtie \pi_n(r) = r.$$

- It is difficult to envision a situation in which a lossy decomposition would be acceptable.

To repair the algorithm for 3NF decomposition, the following step is added.

5. If none of the resulting schemata contains a key of the original relation, create one additional relation which consists of such a (minimal) key.

To repair the example on the previous slide, add a key. It is easy to see that AC is the only candidate key, so add the relation schema R_2[AC]. The old schema R_2[C] may be removed, since it is subsumed by R_2[AC].

General Ideas Regarding Lossless Decompositions:

Fact: Let R[**A**] be a relation schema, and let $\mathcal{F}$ be a set of FD's on **A**. Let

$$D = \{R_1[\mathbf{A}_1], R_2[\mathbf{A}_2]\}$$

be a decomposition of R[**A**] into two projections. Then D is lossless if and only if at least one of the following conditions is satisfied.

- The FD $\mathbf{A}_1 \cap \mathbf{A}_2 \rightarrow \mathbf{A}_1 \setminus \mathbf{A}_2 \in \mathcal{F}^+$.
- The FD $\mathbf{A}_1 \cap \mathbf{A}_2 \rightarrow \mathbf{A}_2 \setminus \mathbf{A}_1 \in \mathcal{F}^+$.

In words, the common attributes must form a key for at least one of the two relations.

Unfortunately, this condition does not extend to the case of a decomposition into three or more relations.

Example: The relation schema R[ABCD] with FD set $\{A \rightarrow C, B \rightarrow D\}$ has the following lossless decomposition:

R_1[AB] with FD set $\varnothing$.
R_2[ACD] with FD set $\{A \rightarrow C\}$.
R_3[BCD] with FD set $\{B \rightarrow D\}$.

- However, the above conditions are not satisfied for any two of the relations.
- Furthermore it is not the case that joining just two of the projections will yield the original relation, even though all of the attributes are covered.

There is a general theory of lossless decompositions. The highlights:

- Instead of working with FD's, one may work directly with "decomposition dependencies:"
  - multivalued dependencies
  - join dependencies

- Niceness is related to a property called "acyclicity," which roughly corresponds to the idea that losslessness may be verified by checking the underlying schemata "two at a time."

- This "niceness" is related to many other "desirable" properties of database schemata, including efficient query processing and management of distributed databases.

Dependency-preserving BCNF decompositions:

- It is not always possible to decompose an arbitrary relation schema, constrained by FD's, into a BCNF relational schema, while preserving dependencies.

- In this sense, BCNF is not "better" – it cannot always be realized!

Algorithm to realize BCNF (may not be dependency preserving):

Input: A single relation schema R[**A**], together with a set $\mathcal{F}$ of FD's on **A.**

1. Set Decomp $\leftarrow$ {R[**A**]}.

2. While there is an S $\in$ Decomp which is not in BCNF do:
   - Choose an S[**B**] $\in$ Decomp which is not in BCNF.
   - Choose an FD $X \rightarrow Y \in \pi_{\mathbf{B}}(\mathcal{F}^+)$ which is not a "BCNF FD;" *i.e.,* the LHS is not a superkey for S[**B**]. Let $\mathbf{B}_1 = X \cup Y$, and let $\mathbf{B}_2 = \mathbf{B} \setminus Y$.
   - Decompose S[**B**] into $S_1[\mathbf{B}_1]$ and $S_2[\mathbf{B}_2]$.

Just as in the case of the 3NF algorithm:

3. If none of the resulting schemata contains a key of the original relation, create one additional relation which consists of such a (minimal) key.

Example:

Rail_Schedule				
Engineer	Train	Date	Departure	Platform

{Train} → {Departure, Platform}
{Engineer, Date, Departure} → {Train}
{Train, Date} → {Engineer}

- The candidate keys are:
  {Engineer, Date, Departure}
  {Train, Date}

- Thus,
  {Train} → {Departure, Platform}
  is a non-BCNF FD.

The new schema is

Train	Departure	Platform

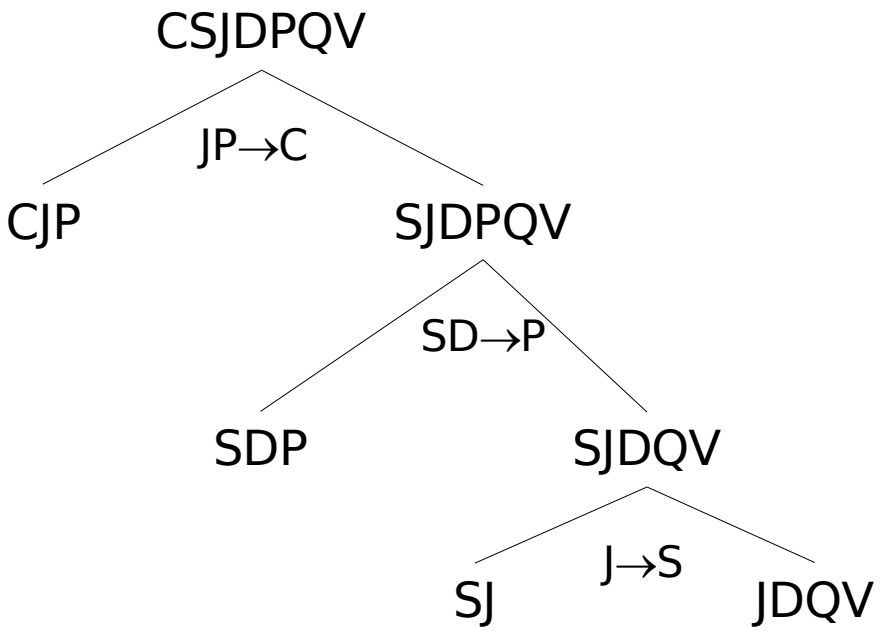
Engineer	Train	Date

Which is the schema obtained earlier. Note that it is not dependency preserving, since
{Engineer, Date, Departure} → {Train}
is lost.

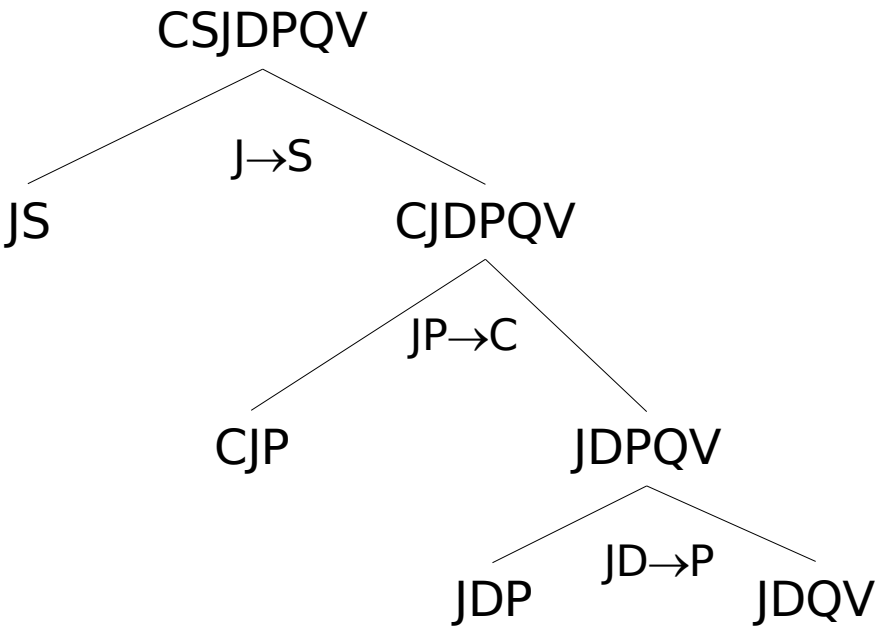
- This schema is in BCNF.

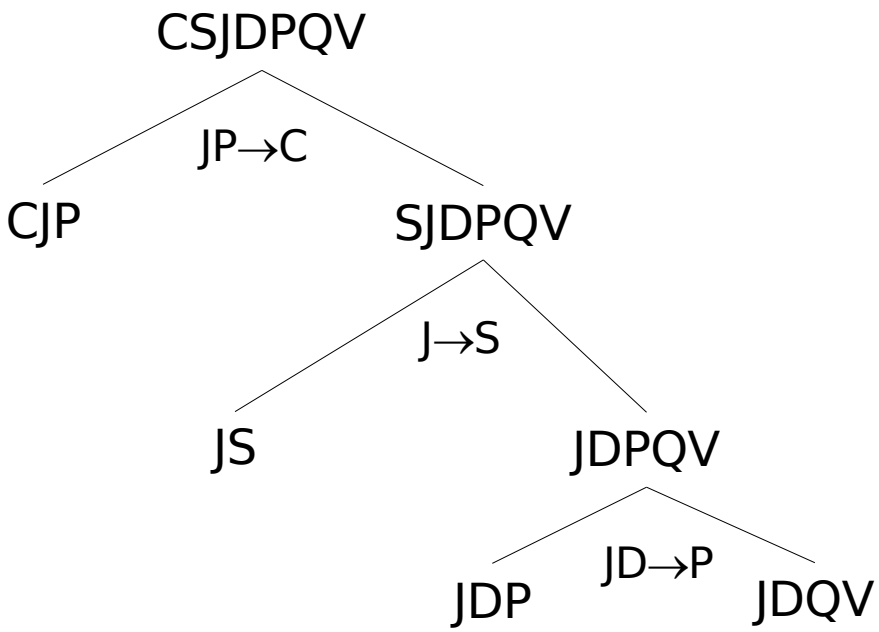
A more complex example:

- R[CSJDPQV] { JP $\rightarrow$ C, SD $\rightarrow$ P, J $\rightarrow$ S }
- Note that { SD $\rightarrow$ P, J $\rightarrow$ S } $\models$ JD $\rightarrow$ P.
- The set of FDs is already minimal.
- Unque key: JDQV
- 3NF decomposition via the algorithm:
  R[JPC], R[SDP], R[JS], R[JDQV]
- To find a BCNF decomposition, it may be necessary to consider several *decomposition trees*:
- The above 3NF is also BCNF, and it is dependency preserving. Here is a decomposition tree:



- The folllowing two decompositions are in BCNF but are not dependency preserving because SD $\rightarrow$ P is not preserved:





Question: Given a single relation schema R[**A**], together with a set $\mathcal{F}$ of FD's on **A** and a minimal cover $\mathcal{G}$ of $\mathcal{F}$, if there is a dependency-preserving BCNF decomposition of this schema, will the 3NF decomposition algorithm always yield a BCNF decomposition?

Answer: No.

Example: R[ABCDE],
$\mathcal{F}$ = { A $\rightarrow$ BC, BC $\rightarrow$ A, BCD $\rightarrow$ E, E $\rightarrow$ C }

- $\mathcal{F}$ is already a minimal cover of itself.

- The 3NF decomposition algorithm yields
  { R[ABC], R[BCDE], R[CE] },
  which is not BCNF since E $\rightarrow$ C is a violating dependency in R[BCDE].

- The alternate decomposition
  { R[ABC], R[ADE], R[CE] }
  is BCNF and dependency preserving.

Reason:

- { A $\rightarrow$ BC, AD $\rightarrow$ E, E $\rightarrow$ C } is an alternative minimal cover for $\mathcal{F}$ which yields this decomposition upon applying the 3NF decomposition algorithm.

- If there is a dependency-preserving BCNF decomposition, it will arise from the 3NF algorithm applied to **some** minimal cover of $\mathcal{F}$.

Final observations on decomposition:

- Note that most commercial DBMS's effectively force BCNF with respect to FD's, because they only allow key constraints.

- Thus, in situations in which no dependency-preserving BCNF decomposition is possible, there will be constraints which are not represented in the schema.

- They also allow foreign keys.

- Using the join tree which represents the decomposition, it is easy to see which foreign-key dependences must be enforced.
  - The common attributes of the local decompositon must be a key for one of the relations.
  - In the other relation, those attributes form a foreign key.

Problems with the projection of FD's:

Here is an interesting example:

R[ABCD] with FD set

$$\mathcal{F} = \{A \rightarrow D,\ B \rightarrow D,\ CD \rightarrow A\}.$$

The view is the projection onto ABC: R_1[ABC].

Facts:

- There is no set of FD's which expresses the constraints on R_1[ABC].
- For any integer n, there is a relation r_1 on attributes AB, containing exactly n tuples, with the property that it is not a projection of any relation

  $$r \in \text{Sat}(R[ABCD], \mathcal{F})$$

  yet any relation obtained by deleting one or more tuples from r is a projection of a legal relation in Sat(R[ABC], $\mathcal{F}$).
- Thus, for no n is it "n-easy" to check the constraints on R_1[ABC].
- The constraints on R_1[ABC] are nonetheless of the $(\forall)..(\forall)$ variety.